Streaming Support for Java RMI in Distributed Environments

Chih-Chieh Yang, Chung-Kai Chen,
Yu-Hao Chang, Kai-Hsin Chung and Jenq-Kuen Lee
Department of Computer Science
National Tsing Hua University
Hsinchu 30013, Taiwan
{ccyang, ckchen, yhchang, kschung}@pllab.cs.nthu.edu.tw, jklee@cs.nthu.edu.tw

ABSTRACT

In this paper we present novel methodologies for enhancing the streaming capabilities of Java RMI. Our streaming support for Java RMI includes the *pushing* mechanism, which allows servers to push data in a streaming fashion to the client site, and the *aggregation* mechanism, which allows the client site to make a single remote invocation to gather data from multiple servers that keep replicas of data streams and aggregate partial data into a complete data stream. In addition, our system also allows the client site to *forward* local data to other clients . Our framework is implemented by extending the Java RMI stub to allow custom designs for streaming buffers and controls, and by providing a continuous buffer for raw data in the transport layer socket. This enhanced framework allows standard Java RMI services to enjoy streaming capabilities. In addition, we propose aggregation algorithms as scheduling methods in such an environment. Preliminary experiments using our framework demonstrate its promising performance in the provision of streaming services in Java RMI layers.

Categories and Subject Descriptors

C.2.4 [**Computer-communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Algorithms, Experimentation, Languages

Keywords

Java RMI, Streaming Java RMI, Aggregation Scheduling Methods, Novel Applications of Java, Java-based Tools

1. INTRODUCTION

The increasing importance of distributed object-oriented environments for use in parallel and distributed service frameworks has increased interest in efficiently supporting for remote-invocation frameworks, since this layer appears to be a promising paradigm for supporting ubiquitous component communications in heterogeneous network environments. This is also the layer where the well-known software layers such as Java Remote Method Invocation (RMI), .NET Remoting, and CCA (Common Component Architecture) remoting are located.

There has been abundant research in this area recently, which has yielded an open RMI implementation that makes better use of the object-oriented features of Java [1]. The ARMI [2] and Manta [3] systems reduce various drawbacks of RMI and provide new RMI-style systems with extended functionality. KaRMI [4] improves the implementation of RMI by exploiting Myrinet hardware features to reduce latencies. A broad range of RMI applications has also been done in [5]. Our research group has also investigated issues associated with the use of RMI in a wireless environment [6, 7]. The Aroma system [8], which is Java-based middleware, aims to exploit Java RMI to replicate objects so as to ensure both availability and adaptability. The dynamic proxy is used as an interceptor to extend the capabilities of Java RMI. In addition, we have also reported on specifications for RMI programs operating in heterogeneous network environments [9]. In recent years, network streaming becomes a highly popular research topic in computer science due to the fact that a large proportion of network traffic is occupied by multimedia streaming. In the network streaming scenario, data that are retrieved from the network can be processed immediately after a sufficient portion (but not necessarily all) of the total stream has arrived. The streaming technology is applied mostly to multimedia data due to the large sizes of these files, allowing users to listen or view a multimedia file while downloading it. Java RMI allows programmers to rapidly establish distributed object-oriented applications. However, the one-to-one and call-and-wait schemes in Java RMI do not match the properties required for streaming in applications. We therefore aimed to extend Java RMI by inserting extra components so as to implement support for streaming.

In this paper we present our novel methodologies for enhancing the streaming capabilities of Java RMI to provide a flexible and convenient framework for streaming applications. Our streaming support for Java RMI includes the *pushing* mechanism, which allows servers to push data in a streaming fashion to the client site via Java RMI, and the *aggregation* mechanism, which allows the client site in a single remote invocation to gather data from multiple servers that keep replicas of data streams and aggregate partial data into a complete data stream. In addition, our system also allows the client site to *forward* local data for further services. Our framework is implemented by extending the Java RMI stub to allow custom designs for streaming buffers and controls, and by provid-

PPPJ 2006, August 30–September 01, 2006, Mannheim, Germany.

ing a continuous buffer for raw data in the transport layer socket. This enhanced framework allows standard Java RMI services to enjoy streaming capabilities. In addition, we propose aggregation algorithms as scheduling methods in such an environment. Our scheme schedules communications by also considering both the resources and data available at each server. Preliminary experiments using our framework demonstrate its promising performance in the provision of streaming services in Java RMI layers.

The remainder of this paper is organized as follows. Section 2 introduces the basics of Java RMI, and Section 3 introduces the software architecture of streaming RMI. Sections 4, 5, and 6 discuss the three important mechanisms in our framework. Section 7 introduces the user APIs of streaming RMI. The results of our performance evaluation are presented in Section 8, several related works are mentioned in Section 9, and conclusions are drawn in Section 10.

2. THE BASICS OF JAVA RMI

In a distributed system, applications run in different machines in the network. These applications need to communicate with each other to send commands and exchange computation results from time to time. A socket is the basic communication mechanism that is supported in almost every programming language and operating system. Sockets are highly flexible, but they are at a level that is too low for developers because new protocols need to be designed for specific purposes, and they need to encode and decode messages for exchange.

The Java platform provides developers with Java RMI, which utilizes features such as dynamic class loading and reflection of Java, and uses a stub object to manage the method invocations. Developers simply create a stub that is used to make method invocations, which takes care of marshalling the invocation and sending it to the remote RMI server. From the developer's viewpoint, this is just like calling a local object.

The architecture of Java RMI is shown in Figure 1.

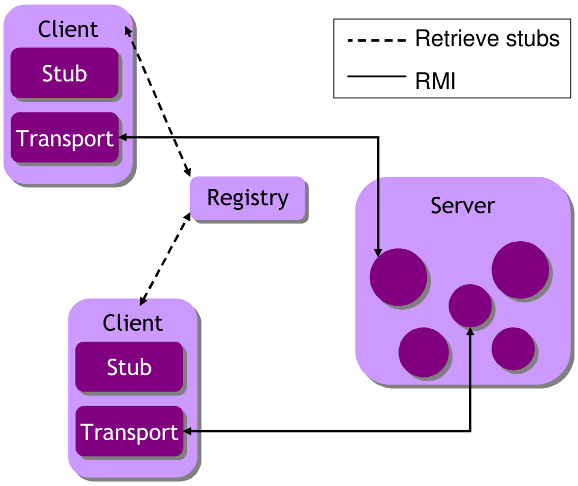

Figure 1: Java RMI architecture.

All RMI servers register with the registry server of their hosting remote objects . RMI clients first look up stubs of a remote object on the registry server, which they subsequently use to call their corresponding remote objects. The stub is very important in Java RMI. It acts as the remote object to the client application by implementing the same interface as the remote object. The client application invokes methods on the stub, and the stub will carry out the methods call on the remote object. It marshals the parameters to the remote JVM, waits for the result, unmarshals the return value, and returns the result to the application. In this work we focus on how to modify the behavior of an RMI stub to integrate streaming ability into Java RMI.

3. SOFTWARE ARCHITECTURE OF STREAMING RMI

The main work of this research was to design an extended architecture for Java RMI to provide capabilities for streaming applications. Figure 2 presents an overview of our proposed architecture, called *streaming RMI*.

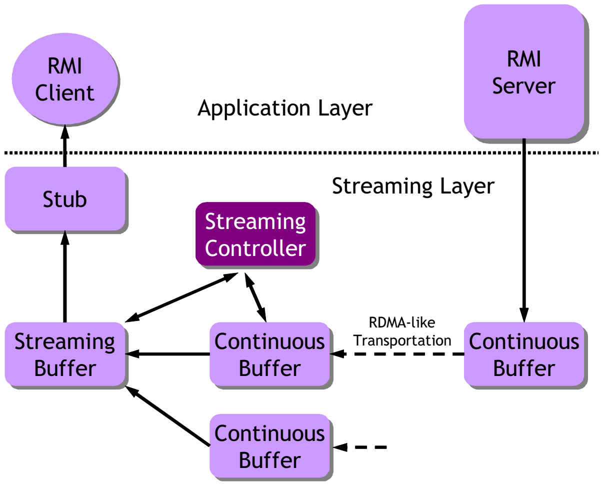

Figure 2: Overview of streaming RMI architecture.

Several important components are needed to support streaming. First, a *continuous buffer* provides the basic buffering of component communications. When a server wants to send data to a client, it writes its own continuous buffer that will push the data through the network to the client. A client reads continuous buffers of its own to receive data from the network. Second, there are two additional components needed in the client-side architecture: *streaming controller* and *streaming buffer*. The streaming controller is responsible for scheduling how servers should send their data, and for aggregating the received data from continuous buffers. The streaming buffer is a repository for aggregated data. With these components, a streaming RMI client can gather data from many different streaming RMI servers simultaneously and aggregate the data received into a new data stream . Figure 3 shows how these new components can be deployed in the layers of standard Java RMI.

Figure 3 extends the Java RMI framework on both the client and server sides. In the client side, we extend the stub to include a streaming controller and a streaming buffer. We also extend the transport layer socket to include continuous buffer support. In the server side, we extend the stub with a loader and a continuous buffer. The detailed behavior of each component is as follows:

Stub The stub of Java RMI is usually generated automatically. In order to add customized features, we disassemble the class file generated by the RMI compiler (`rmic`) to Java source code, and modify its content to intercept method calls. We make it able to create multiple streaming sessions to different servers, and able to pass intercepted method invocations to

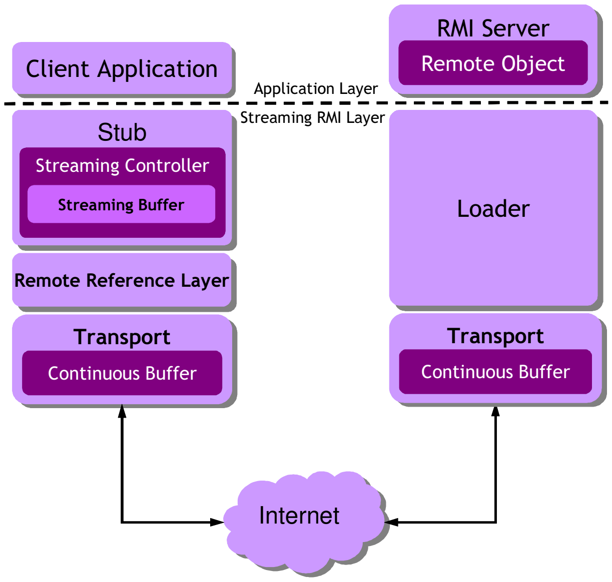

Figure 3: Details of streaming RMI architecture.

the streaming controller that is responsible for parsing their meaning and making necessary modifications before sending the invocations to streaming RMI servers.

Streaming Controller The streaming controller communicates with the stub. It receives remote invocations intercepted by the stub and parses their content. After parsing, the stub may make modifications, distribute the invocations to many streaming RMI servers, or directly return results that already reside in the local streaming buffer. It is also responsible for scheduling which data each server should send to clients, and how this should be achieved. The streaming controller also gathers raw data received in continuous buffers, aggregates them, and stores them in a streaming buffer.

Streaming Buffer The streaming buffer exists only on the client side of a streaming RMI infrastructure, and serves as an aggregation repository. A streaming RMI client may receive raw data from different servers in the network. The streaming controller knows how to aggregate these raw data and store the results in the streaming buffer for applications to retrieve.

Continuous Buffer The continuous buffer is responsible for pushing raw data from the server side to the client side, and so it exists both on the client and server sides and can be treated as a pair. The continuous buffer is in charge of buffering data to send and sending data to the corresponding continuous buffer on the client side at a specific rate. In addition, the upper layer of the client side is able to read data from the local continuous buffer for further processing.

Loader Standard Java RMI on the server side is designed to passively respond to client requests. To maximize efficiency, streaming RMI servers should be able to actively send data to clients. To achieve this, the loader is added to the server-side architecture to periodically load streaming data into the continuous buffer after an initial streaming session is set up. The data in the continuous buffer will then be pushed to clients.

A streaming RMI session is started as follows:

1. The server of the streaming session begins listening for an incoming connection request.
2. A client queries for the specific type of remote object with an RMI registry to obtain a stub.
3. The client application calls a specific initialization method on a stub. The stub will forward the invocation to the streaming controller.
4. The streaming controller parses the method invocation, looks for available streaming RMI servers, creates continuous buffers, and schedules how servers should send streaming data.
5. Initialization messages are sent to streaming RMI servers, as in standard RMI; and continuous buffers create extra connections to server-side continuous buffers to support data pushing (see Section 4).
6. Data are loaded by server-side loaders to continuous buffers and pushed to the client through sockets automatically. The streaming controller collects and aggregates data, which are stored in the streaming buffer after aggregation.
7. If subsequent data requests from client applications can be satisfied by data stored in the streaming buffer, these method invocations will be returned immediately.

4. PUSHING MECHANISM

This section describes pushing, which is the data transmission mechanism in our streaming RMI architecture that replaces the call-and-wait behavior in standard Java RMI. Our automatic mechanism of pushing from a server to clients is achieved by creating a dedicated communication channel for data pushing, with the original channel used by RMI to exchange control messages only. The loader in the server of the streaming RMI infrastructure creates a server socket and listens for incoming connections from clients. In order to serve multiple clients, the loader creates a new thread and activates the corresponding continuous buffer to handle the pushing mechanism of the client each time it accepts a client socket. The client sends the client ID through the socket to the server, allowing the server to identify this client and know where to push the data. After these session setup steps, the continuous buffer of the streaming RMI server starts pushing data to the client.

Application of the pushing mechanism requires a method to coordinate the pushing speed of the server. Because the physical buffer length is always limited, the server may overwrite data not yet received by the client if the data rate is too high, which can cause a fatal system error.

Three flow control policies of the pushing mechanism are considered in our design:

1. Send Only When Consumed: In this scheme the client sends the state of the buffer pointer to the server to determine how much data has been consumed by client applications. The server then pushes this amount of data to the client.
2. Adaptive: In this scheme the servers dynamically adjust the pushing rate according to the buffer status of the client. The adaptation is triggered when the client-side continuous buffer is empty or full for a certain time period. The server will then increase or decrease the pushing speed accordingly.
3. As Fast As Possible: In this scheme the servers push data as fast as possible. This may cause some of the data in client buffers to be overwritten. This policy can be applied when

the data arrival speed is more important than data integrity or when expired data are useless to the user applications.

The application developers can choose the most suitable flow control policy for specific applications, which makes the use of streaming RMI more flexible in various distributed applications.

5. AGGREGATION

When a streaming RMI client requires a specific data stream, there may be multiple streaming servers that are able to supply it. It is likely to be desirable to create multiple connections to these servers for several reasons: (1) this will share the load among the servers (load balancing), (2) the bandwidth from several servers can be used to supply the stream simultaneously so as to ensure smooth playback, and (3) different data streams from different servers can be merged into a new aggregated stream. Information on the availability of streaming servers can be obtained in several ways. For example, one can use a peer-to-peer query (if we choose peer-to-peer environment as our network transport layer) or via a centralized content-locator server. In our current implementation for experiments, we adopt the second approach. A peer-to-peer environment can can be incorporated as well. Information of the streaming servers containing specific content is obtained by the streaming controller of a streaming RMI client simply querying the content locator. After the information of available streaming servers is available, a schedule on how these servers should send the stream needs to be made. The optimal schedule depends on the types of streams we retrieve, examples of which include the same static (fixed-sized) content spread across different servers (such as a movie) and different streams with the same properties (such as video and audio streams of a baseball game) that will be aggregated locally. Our streaming RMI framework makes it possible for developers to customize the scheduling policies that are best for their streams with component interfaces. After scheduling, the schedules are sent out through multiple RMI connections to all of the associated servers.

In this work, we focused on how to gather static data streams from multiple servers that keep replicas of streams and aggregate partial data into a complete data stream. We first introduce some notation used to describe our aggregation algorithm:

- A set of streaming servers: $S = \{s_i | i = 1, \ldots, n\}$.
- A set of data blocks: $D = \{d_j | j = 1, \ldots, m\}$.
- For each streaming server s_i:
  - The supplying bandwidth b_i of s_i.
  - A set of data blocks that exists in s_i: $Blocks(s_i)$.
  - The completeness of data in s_i: $Completeness(s_i)$
  - The amount of content: k_i
- The bandwidth requirement of the streaming session: $Req(d_j)$
- The bandwidth allocation table: $BAT_{m \times n}$

Algorithm 1 presents a mechanism to schedule the same data stream coming from different servers. First, we evaluate the *weight* for each streaming server, which represents their priorities. A server with a higher weight is said to be *preferred.* Three factors are used to evaluate the weight of a server. If the bandwidth of a server is higher, the server is capable of supplying more streaming sessions. Another factor is the completeness of the data stream – when a server has most of a data stream, it is more preferred. It is because

Algorithm 1: Bandwidth allocation for aggregation

```
begin
    /* Evaluate weight of each server    */
    foreach streaming server s_i in S do
        Weight(s_i) =
        α × b_i/Req(d_j) + β × Completeness(s_i) + γ × (1/k_i)
    end
    /* Sort list S by the weight of each
       server in decreasing order        */
    Sort(List S)
    /* Allocate bandwidth                */
    for i=1 to n do
        for j=1 to m do
            if Req(d_j) > 0 and d_j in Blocks(s_i) then
                BAT(i, j) = MIN(b_i, Req(d_j))
                Req(d_j) = Req(d_j) − BAT(i, j)
            end
        end
        if every element in Req is 0 then
            Break
        end
    end
end
```

Server ID	Bandwidth	Set of blocks contained	Amount of content
A	10	{0, 1, 2, 3}	10
B	20	{0, 1}	5
C	10	{2, 3}	10

Table 1: A running example for Algorithm 1.

when we allocate supplying bandwidth from a streaming server, the bandwidth will be occupied by the requesting client from the time of admission till the end of the session. If it is only a small portion of the complete data stream, there must be a lot of idle time for the pushing thread of the streaming server, and the allocated bandwidth is wasted. The completeness is by

$$Completeness(s_i) = \frac{size(Blocks(s_i))}{size(D)}, \tag{1}$$

The third factor we considered is the amount of different content on a streaming server. A server is less preferred if it has a larger amount of different data streams, because it is better to first utilize the bandwidth of those with fewer data streams, as they offer less flexibility in the scheduling process. The server that is harder to utilize is given work first whenever it has data that are required. The weight of a server is defined as

$$Weight(s_i) = \alpha \times b_i + \beta \times Completeness(s_i) + \gamma \times \frac{1}{k_i}, \tag{2}$$

where α, β, and γ are coefficients that vary among network environments and parameters.

However, the α in equation (2) which falls in the range from 0 to b_i, might effect on the result weight too much compared with other coefficients. Normalization helps us better choose appropriate coefficients. The normalized weight equation is defined as

$$Weight(s_i) = \alpha \times \frac{b_i}{Req(d_j)} + \beta \times Completeness(s_i) + \gamma \times \frac{1}{k_i}, \tag{3}$$

In Equation (3) we fixed the α coefficient by letting it be divided by the required bandwidth of the data stream. Therefore, we can set the α, β, and γ coefficients from 0 to 1 to calculate the weight.

The input to Algorithm 1 is the set of streaming servers, S. We first evaluate weights for each server, and then sort the list by weight. Next, we use up all of the bandwidth that the topmost server can provide, followed by the second, etc., until all the required bandwidth is satisfied. The experiments described in Section 8 demonstrate the effects of our proposed scheme.

Table 1 lists an example to illustrate how our algorithm works. A stream with four blocks $\{0, 1, 2, 3\}$ is requested, the bandwidth for each block is 20 and there are three available streaming servers containing blocks. If we choose α=0.1, β=0.5, and γ=1, equation (3) indicates that the weight of server A is $0.05+0.5+0.1 = 0.65$, the weight of server B is $0.1 + 0.25 + 0.2 = 0.55$, and the weight of server C is $0.05 + 0.25 + 0.1 = 0.4$. Server A has the highest priority, so we schedule it first. We allocate all the bandwidth needed by blocks $\{0, 1, 2, 3\}$ to it. After that, we allocate the bandwidth needed by blocks $\{0, 1\}$ from server B. If the bandwidth requirement is not satisfied by servers A and B, we allocate bandwidth from server C. If the requirement cannot be satisfied by these three streaming servers, the system reports that the allocation operation has failed.

6. FORWARDING

It is possible that some popular media streams will be downloaded by many requesting clients simultaneously. In this situation, the loading of streaming servers can be reduced if some of the active clients are able to forward their content to other clients. For example, in Figure 4, requesting clients R_2 and R_3 can obtain the data stream from forwarding client R_1.

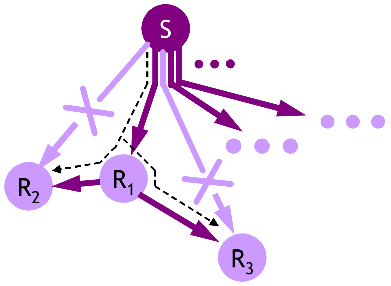

Figure 4: Forwarding behavior.

In our design a requesting client can query the content locator for available forwarding clients, which should reply with several parameters, including the set of data blocks currently in its buffer, the time to live (TTL) for the set of blocks (i.e., the time at which the set will be available), and the bandwidth it can supply. We give the descriptions for those parameters below.

Set of Forwardable Data Blocks The forwarding client may have some blocks buffered before they are consumed by applications. These blocks can be forwarded to other clients, and hence the forwarding client should notify the content locator with the number of blocks available. We assume that the forwarding client is able to forward all subsequent blocks of the stream.

Time To Live We need an estimate of the time at which these blocks will exist in the buffer before they expire; this information is provided to requesting clients.

Supplying Bandwidth The forwarding client also needs to report the amount of bandwidth it has available.

Algorithm 2: Looking for forwarding clients

```
/* Query the content locator about the
   availability of forwarding clients    */
begin
    foreach response from forwarding clients do
        foreach block in the set of forwardable blocks do
            if the TTL of this block is acceptable then
                Treat this forwarding client as a streaming
                server containing all the required blocks after
                this one
                Break
            end
        end
    end
end
```

Algorithm 2 presents our algorithm for the forwarding case. When a requesting client needs forwarding clients, it sends a query to the content locator as for querying a streaming server. When it receives the responses, it has to check the TTLs of the set of data blocks and dump those that may be expired when the client needs them. If a block passes the test, the requesting client can assume the forwarding client is able to forward all subsequent blocks of the specific stream, and schedule them as resources according to Algorithm 1.

7. SOFTWARE APIS OF STREAMING RMI

The major advantage of the Java RMI programming model is that it hides the details of network communications. Our intention here is to add streaming ability to the model while preserving this advantage. In addition, we provide developers with as much flexibility as possible in manipulating various types of data stream. Developers are able to specify their customized scheduling policies to optimize the performance and aggregation operators for post-processing.

The code shown in Figure 5 illustrates how to write a streaming RMI client, which was actually used in our experiment. (See Section 8.) It simply reads the remote object for specific amount of data and writes the data read into a file. As shown in the example, the initialization steps, such as registry creation and looking up for the stub, are similar to standard RMI. The difference is in the line that `dataObj.initialize()` is invoked, where several streaming session setting parameters (including the address of the content locator, streaming data ID) are wrapped in object `InitSetting` and sent through standard RMI transport channel to participating servers. The parameters are received by the loader at the server for session setup procedures. After these initialization steps, the client simply calls `dataObj.read()` to copy data to its own buffer. The streaming server code is not listed here, because it is almost exactly the same as a standard RMI server program except for the use of the `StreamingData` class as the type of remote object.

We are able to specify customized scheduling policies by specifying the type of scheduler to load in the configuration file (if this is not done, the default scheduler will be applied). The interfaces of the scheduler and operator are listed in Figure 6. A scheduler takes a set of available sources and constraints of the streaming session as input and returns a set of schedule plans once scheduling is completed.

Figure 6 also defines the interface of aggregation operators. Programmers need to implement the `Process()` method, which takes a set of data blocks from different continuous buffers as input and outputs a set of processed data blocks. The use of the two methods `SetNextOperator()` and `GetNextOperator()` allows op-

```
public class StreamingClient {
  public static void main(String args[]) {
    byte[] buf = new byte[100 * 1024]; //read buffer
    int toRead = 10*1024; // read block size = 10kB
    int bytesRead = 0;
    try {
      Registry registry =
        LocateRegistry.getRegistry(host, 9999);
      StreamingInterface dataObj =
        (StreamingInterface)
        registry.lookup("StreamingData");
      String id =
        dataObj.initialize(
        new InitSetting(host, "test.avi")
        );
      File file = new File("copy.avi");
      FileOutputStream os = new FileOutputStream(file);
      while (true) {
        bytesRead = dataObj.read(buf, 0, toRead);
        if (bytesRead == -1) {
          os.close();
          return;
        }
        os.write(buf, 0, bytesRead);
        if (bytesRead == 0)
          Thread.sleep(5); \\ rest for a while
        }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Figure 5: Code listing: initialization of a streaming RMI client.

```
public IScheduler {
  Plan[] Schedule(Node[] sources, Constraints c);
}

public IOperator {
  Block[] Process(Block[] blocks);
  void SetNextOperator(IOperator op);
  IOperator GetNextOperator();
}
```

Figure 6: Code listing: interfaces of scheduler and operator.

erators to be chained to each other to perform a series of operations. For example, we can chain a decompression operator and a merge operator to decompress streams before merging them.

The above examples indicate that creating a streaming RMI application is as simple as creating a standard RMI one. Our streaming RMI framework also provides flexibility for developers to perform various aggregation operations and to apply customized scheduling policies.

8. EXPERIMENTS

We evaluated the performance and overhead of our streaming designs using experiments in which it was deployed in a simple streaming application, and compared it with standard RMI. In addition, we performed a simulation to show the advantages of our proposed aggregation algorithm.

In our first experiment, we evaluated the performance by setting up a streaming server and a requesting client (which is listed in Section 7) over a 100Mbits/sec bandwidth link. The programs are built and tested under JDK version 1.5.0_04 in Microsoft Windows XP machines. The client tried to retrieve a data stream by repeatedly requesting a fixed-sized block. We measured the throughput obtained by varying the block size the client requested in each `read` operation. In streaming RMI, the operation should benefit from our pushing mechanism for its prefetching behavior.

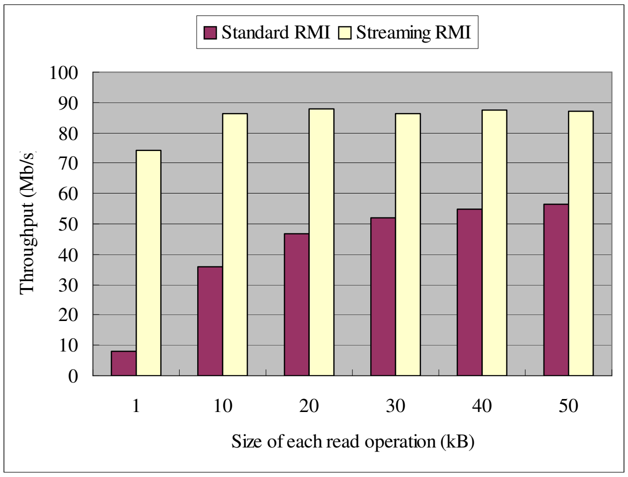

Figure 7: Performance benefits with the pushing mechanism.

Figure 7 presents the performance comparison, and indicates that the smaller the read block size was, the worse standard RMI performed. From the result, we observed that pushing mechanism actually made data streaming effective, because streaming RMI was able to utilize about 85% of network bandwidth provided, while the throughput of standard RMI suffered from frequent remote method invocations to the RMI server. For streaming RMI, method invocations are performed only at initialization or changing streaming settings and data are automatically pushed from server to client without extra remote method invocations, while standard RMI needs client to make a remote method invocation with streaming server for each block requested. Although the situation for standard RMI improved with the increasing size of read blocks, its throughput was still far behind the throughput of streaming RMI.

Read block size (kbyte) Read times	1 5000	10 500	50 100	100 50
Standard RMI read overhead (byte)	243777	25000	5979	3129
Streaming RMI read overhead (byte)	769	768	769	769
Standard RMI write overhead (byte)	228798	23413	5223	2880
Streaming RMI write overhead (byte)	647	646	647	646

Table 2: The data overhead in sum caused by invocation messages.

In the second experiment, we compared the additional data overhead caused by invocation messages of standard RMI with the overhead in streaming RMI. We set up a streaming server and a client. The client requested for a 5 MB data stream from the server by repeatedly reading a fixed-sized block, and we used a counting socket in RMI transport layer to measure total amount of bytes sent and received from the socket. In standard RMI, each `read` operation is marshalled by its stub and sent to RMI transport layer, but in streaming RMI, the `read` operation performed by a client is handled by *streaming controller* by checking the local continuous buffer for data prefetched from a dedicated pushing channel. As a result, these invocations are handled locally without travelling through the network. We suppose the data overhead from invocation messages is greatly reduced in streaming RMI, and the experimental results proved it.

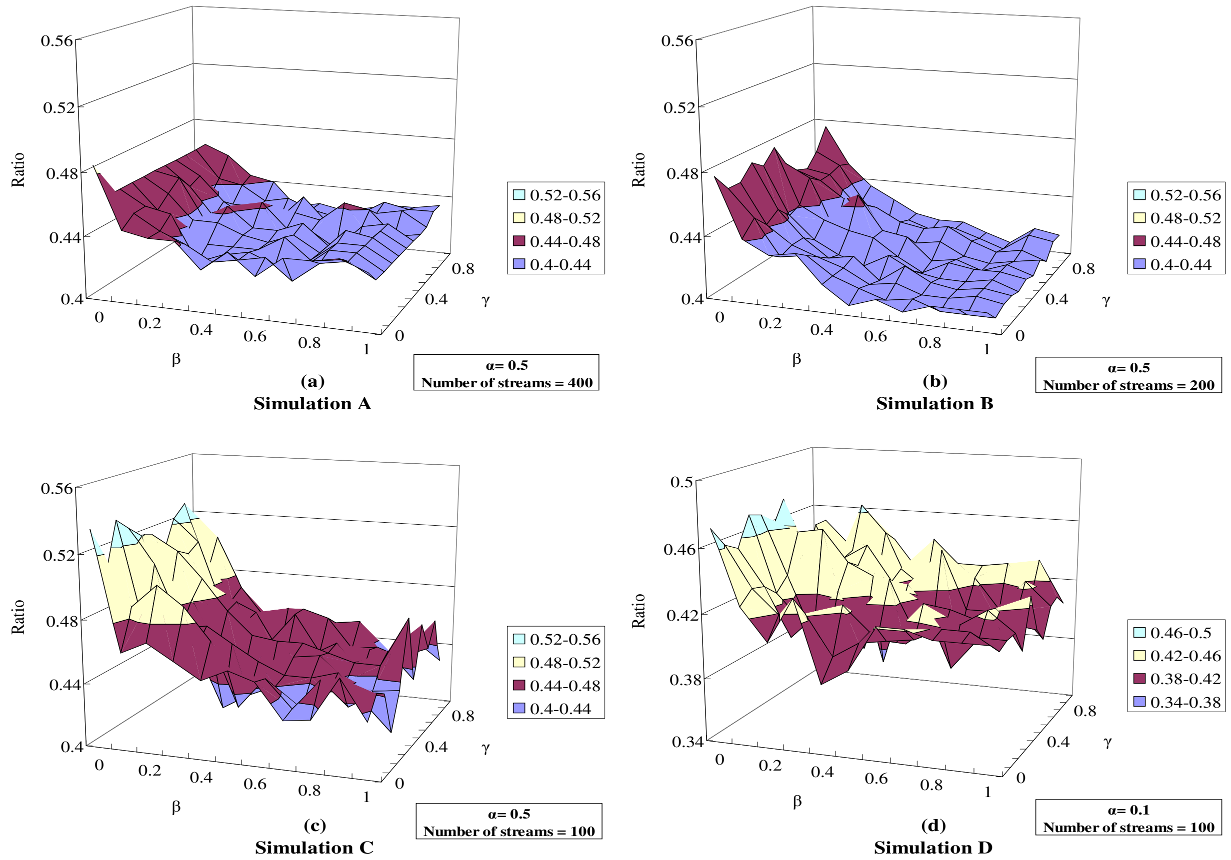

Figure 8: The simulation results with different coefficient weights from our aggregation method.

From Table 2, we are able to see that in standard RMI, the total data overhead was large when the read block size was small. Because the client needed to send more invocation messages for the same stream. With pushing mechanism, the data was prefetched in the local buffer of the client, and the need of exchanging invocation request and response messages was greatly reduced. Thus, no matter which size of the read block was, the total overhead was nearly the same and minimal.

Next, we designed an experimental simulator for investigating the effects of our proposed aggregation policy. We input parameters needed into the simulator, and the simulator runs by the following steps:

1. Creates nodes and bandwidth information.
2. Generates and distributes streaming contents into the nodes.
3. Generates and distributes streaming requests into the nodes.
4. Run the simulation. Scheduling is performed at the time each request is issued.

Table 3 lists the parameters and their values used in the simulation. The number of nodes, the number of different streams, the simulation period, and the size of the data stream were fixed. The number of duplicates of a stream, the bandwidth settings of nodes, and the bandwidth requirements of streams were selected considering behaviors of real life applications, with the samples being uniformly distributed between the upper and lower bounds. We tested the performance under different coefficient sets with a fixed requests. In our simulation, parameters were set as shown in Table 3. The results are presented in Figure 8.

Parameter	Value
Number of nodes	1000
Size of data streams (MB)	2
Simulation period (s)	3600
Number of duplicates	1, 2, 3, . . . , 100
In-bound bandwidth (kB/s)	8, 16, 32, 64, 128
Out-bound bandwidth (kB/s)	8, 16, 32
Bandwidth requirement (kB/s)	8, 16, 32, 64, 128, 256, 512
Number of requests	5000
β, γ	0, 0.1, 0.2, 0.3,. . . , 1.0

Simulation	(a)	(b)	(c)	(d)
α	0.5	0.5	0.5	0.1
Number of different streams	400	200	100	100

Table 3: Parameter sets used in the simulation.

The x-axis is β, the y-axis is γ, and the z-axis represents *the ratio of average waiting time* of systems deploying our aggregation policy compared with those without aggregation policy (which means that only nodes with complete data streams are able to supply the request). Thus the the lower z values represents the better results.

In Figures 8(a), 8(b), and 8(c), there is a clear trend that when the coefficient β (which represents completeness of a data stream) was increased, the ratio of average waiting time is lower. This shows the importance of β in our aggregation policy. Also by observing the three figures, we know that when the number of different streams reduced, the slope was getting steeper but the results are not overall worse. It indicates the possiblity for a system manager to adjust the coefficients to get better performance, or the system automatically adjusts them from machine learning experiences when stream sources become fewer.

And from Figures 8(c) and 8(d), we found that when α (which represents the importance of available bandwidth) was lower, the effect of β and γ is magnified, and achieved overall better results. We suppose that with aggregation, the bandwidth requirement of a streaming session is easy to be satisfied while there are lots of duplicates in the network. Thus the importance of α is reduced.

The simulation results indicate that our aggregation policy greatly reduces the average waiting time, and that better performance is obtained in all cases when all three factors are considered. It also demonstrates that the completeness of a stream is the most important factor, and suggests that tuning of coefficients may further improve system performances.

9. RELATED WORK AND DISCUSSION

Streaming multimedia has great increased in popularity in recent years, in applications such as video on demand. A streaming module implements "play while downloading" instead of "play after downloading". Several techniques have been proposed to implement streaming on the Internet. A distributed video streaming is presented in [12], where each session involves multiple replicated video servers. The idea is similar to our aggregation mechanism in Section 5. However, the system lacks scalability without mechanisms like forwarding, so the system may be overloaded if the number of clients grows. In [13], multiple reliable servers are installed across the network to act as a software router with multicast capability. This approach can allow a client to obtain data not only from the source but also from the software router, thus alleviating the bandwidth demand on the source, which is similar to the effect of the forwarding mechanism.

The Object Management Group has defined a specification for the control and management of audiovisual streams [14], based on the CORBA (Common Object Request Broker Architecture) reference model [15]. This specification defines the mode of implementation of an open distributed multimedia streaming framework. In [16], TAO audiovisual streaming services are addressed and provide applications with the advantages of using CORBA IIOP in their stream establishment and control module, while allowing the use of more efficient transport-layer protocols for data streaming. An explicit open bindings mechanism is proposed in [17], which allows the programmer to set up an additional transport connection between CORBA objects. This connection can be used for more precise control over streaming session. This is similar to our work in extending a remote method invocation mechanism to provide streaming ability. Nevertheless, it only focuses on the data delivery process, and lacks data aggregation mechanism for users to better manipulate the content of streaming data.

Compared with previous research works, our work is designed to extend the streaming capabilities for standard Java RMI, which is a well known language feature of a popular object-oriented programming language. We also provide software APIs for programmers to build streaming applications on top of our framework.

In this paper we provide an alternative method on handling streaming data. We model data streams into objects which can be accessed remotely by our extended RMI mechanism. Our design also presents a framework allowing the data stream objects to distribute the delivering jobs to other duplicated source and the client to schedule and aggregate the data. Different scheduling algorithms can be plugged in this framework to provide adequate functionality. We believe this extended RMI can be used as prime functions exposed to the users of Data Stream Management Systems (DSMS) such as Aurora [18] and Nile [19]. In JSR 158: Java Stream Assembly [20], a model for stream manipulation is proposed, which is similar with what we would like to achieve in aggregation mechanism. We might consider the possibilities of integrating these techniques into our framework in the future.

As to the issue of garbage collection, the processing and delivering of streaming data might be interfered when garbage collection occurs. In our implementation we reuse most objects such as temporary buffers to reduce the need of heap allocation. Nevertheless this can not completely avoid garbage collection. To guarantee smooth streaming processing, a system incorporating performance issues and models of real-time garbage collection schemes is also be needed.

10. CONCLUSIONS

In this paper we present our novel methodologies for enhancing the streaming capabilities of Java RMI by inserting additional components such as streaming controller, streaming buffer, continuous buffer, and loader. These components are designed for supporting core streaming technologies with pushing, aggregation, and forwarding mechanisms. Our work also comes with a software design that is ready for use. Preliminary experiments performed using our framework demonstrate the promising performance of our proposed schemes in providing streaming services in Java RMI layers.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable suggestions and comments that have helped to improve the presentation and quality of our paper.

This work was supported in part by Ministry of Economic Affairs under grant no. 95-EC-17-A-01-S1-034, by National Science Council under grant no. NSC 94-2220-E-007-019, NSC 94-2220-E-007-020, NSC 94-2213-E-007-074, NSC 95-2752-E-007-004-PAE, and NSC 95-3113-P-007-002, and by Institute for Information Industry under grant no. 95-EC-17-A-99-R1-0461.

11. REFERENCES

[1] G. K. Thiruvathukal, L. S. Thomas, and A. T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(11–13):911–925, 1998.

[2] R. R. Raje, J. I. Williams, and M. Boyles. Asynchronous Remote Method Invocation (ARMI) mechanism for Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, 1997.

[3] J. Maassen, R. van Nieuwport, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java remote method invocation. In: *The Proceedings of the 7th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Atlanta, GA, pp. 173–182, May 1999.

[4] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In: *Proceedings of the ACM Java Grande Conference*, San Francisco, CA, pp. 152–157, Jun. 1999.

[5] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI performance and object model interoperability: experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–956, 1998.

[6] P. C. Wey, J. S. Chen, C.-W. Chen, and J.-K. Lee. Support and optimization of Java RMI over Bluetooth environments. In: *Proceedings of the ACM Java Grande - ISCOPE Conference*, Seattle, WA, 17:967–989, Nov. 2002.

[7] C.-W. Chen, C.-K. Chen, J.-C. Chen, C.-T. Ko, J.-K. Lee, H.-W. Lin, and W.-J. Wu. Efficient support of Java RMI over heterogeneous wireless networks. In: *Proceedings of ICC*, Paris, France, pp. 1391–1395, Jun. 2004.

[8] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Interception in the Aroma system. In: *Proceedings of the ACM Java Grande Conference*, San Francisco, CA, pp. 107–115, Jun. 2000.

[9] C.-K. Chen, C.-W. Chen, and J.-K. Lee. Specification and architecture supports for component adaptations on distributed environments. In: *Proceedings of IPDPS*, Santa Fe, NM, pp. 47a, Apr. 2004.

[10] C.-K. Chen, Y.-H. Chang, C.-W. Chen, Y.-T. Chen, C.-C. Yang, and J.-K. Lee. Efficient switching supports of distributed .NET Remoting with network processors. In: *Proceedings of ICPP*, Oslo, Norway, pp. 350–357, Jun. 2005.

[11] National Science Council (NSC). *Research Excellence Project.* http://www.ccrc.nthu.edu.tw/PPAEUII/.

[12] T. Nguyen and A. Zakhor. Distributed video streaming over Internet. In: *Proceedings of SPIE/ACM MMCN*, San Jones, CA, Jan. 2002.

[13] J. Jannotti, D. K. Gifford, and K. L. Johnson. Overcast: reliable multicasting with an overlay network. In: *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, pp. 209–301, Oct. 2000.

[14] Object Management Group. Control and management of A/V streams specification. OMG document telecom, 97-05-07 edn., Oct. 1997.

[15] Object Management Group. The common object request broker: architecture and specification, edn. 2.2, Feb. 1998.

[16] S. Mungee, N. Surendran, D. C. Schmidt. The design and performance of a CORBA audio/video streaming service. In: *Proceedings of the Hawaii International Conference on System Sciences*, Maui, Hawaii, pp. 8043, Jan. 1999.

[17] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin. Supporting adaptive multimedia applications through open bindings. In: *Proceedings of ICCDS*, Annapolis, Maryland, pp. 128, May 1998.

[18] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management In: *The VLDB Journal*, 12(2):120–139, 2003.

[19] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, and X. Xiong. Nile: a query processing engine for data streams. In: *Proceedings of ICDE*, Boston, MA, pp. 851-, Mar 2004.

[20] Sun Microsystems, Inc. JSR 158: Java Stream Assembly. http://jcp.org/en/jsr/detail?id=158.